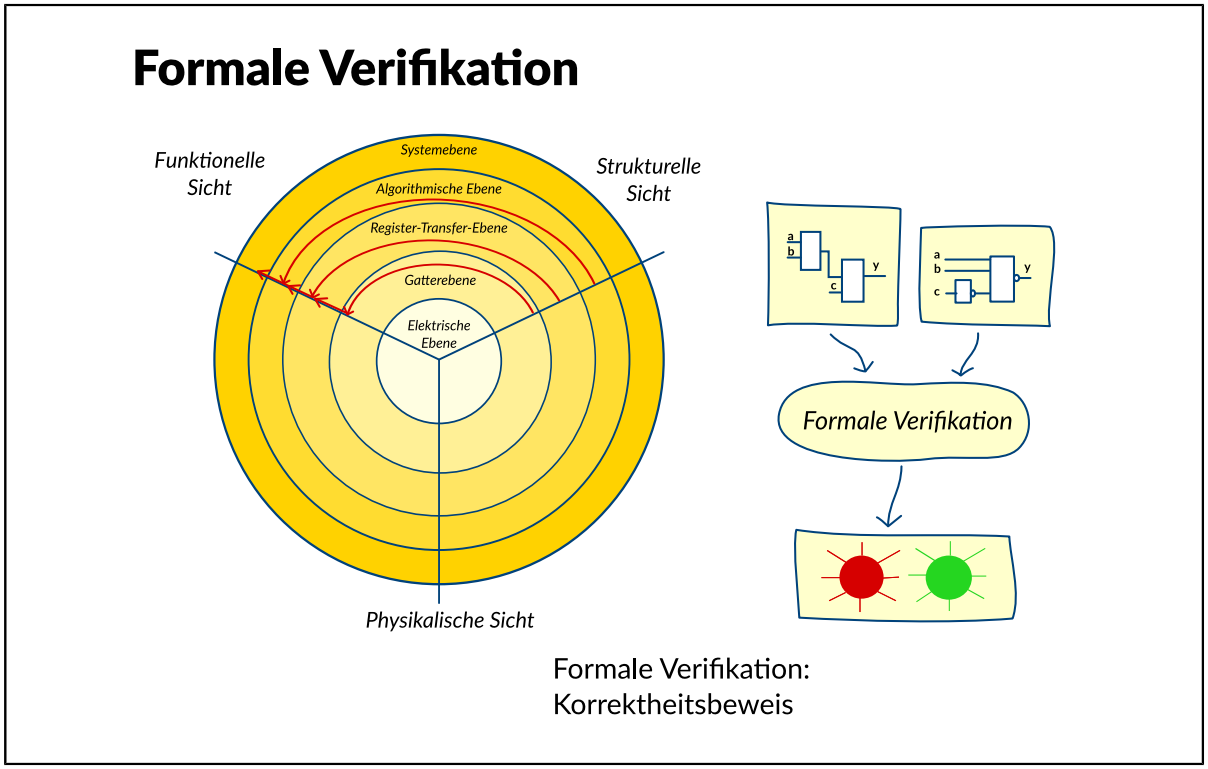
Electronic Design Automation (EDA)

Formale Verifikation

Date of generation: 30.5.2023

Formale Verifikation: Formale Verifikation

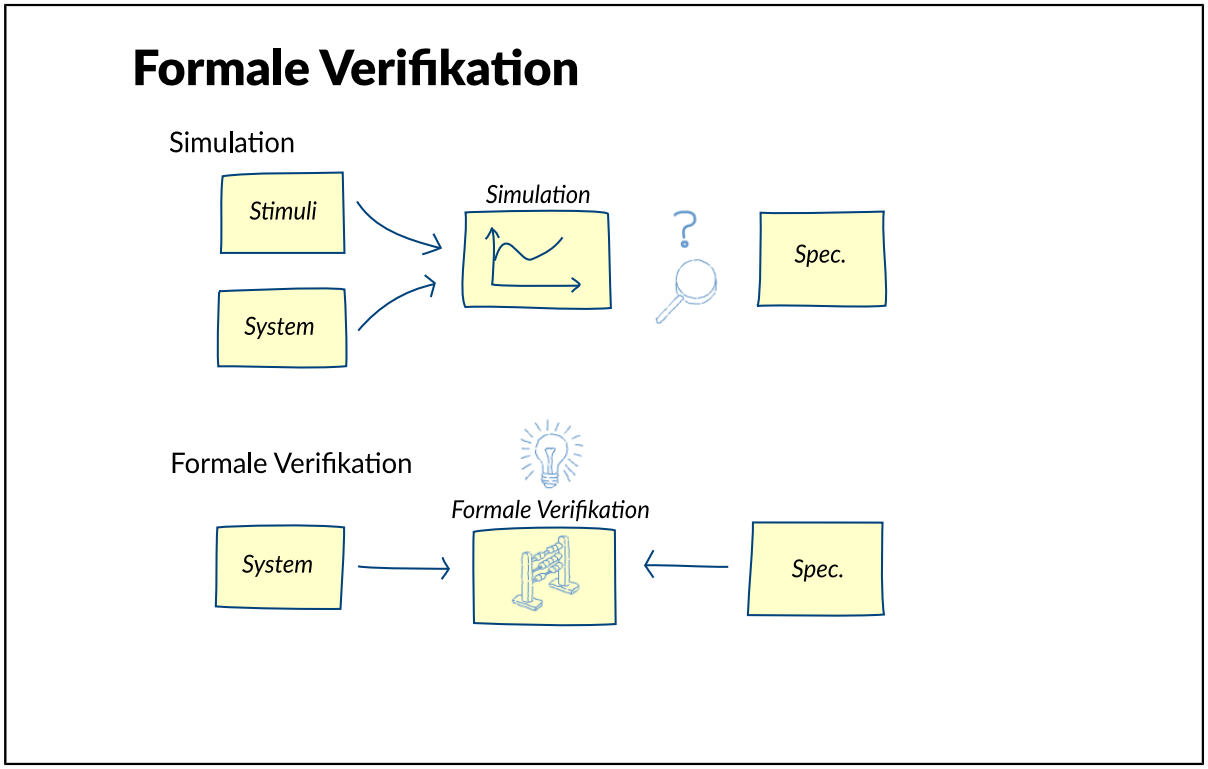


Formale Verifikation:
Korrektheitsbeweis

Unter formaler Verifikation versteht man Verfahren, die die korrekte Funktion eines Systems anhand von mathematischen Methoden nachweisen.

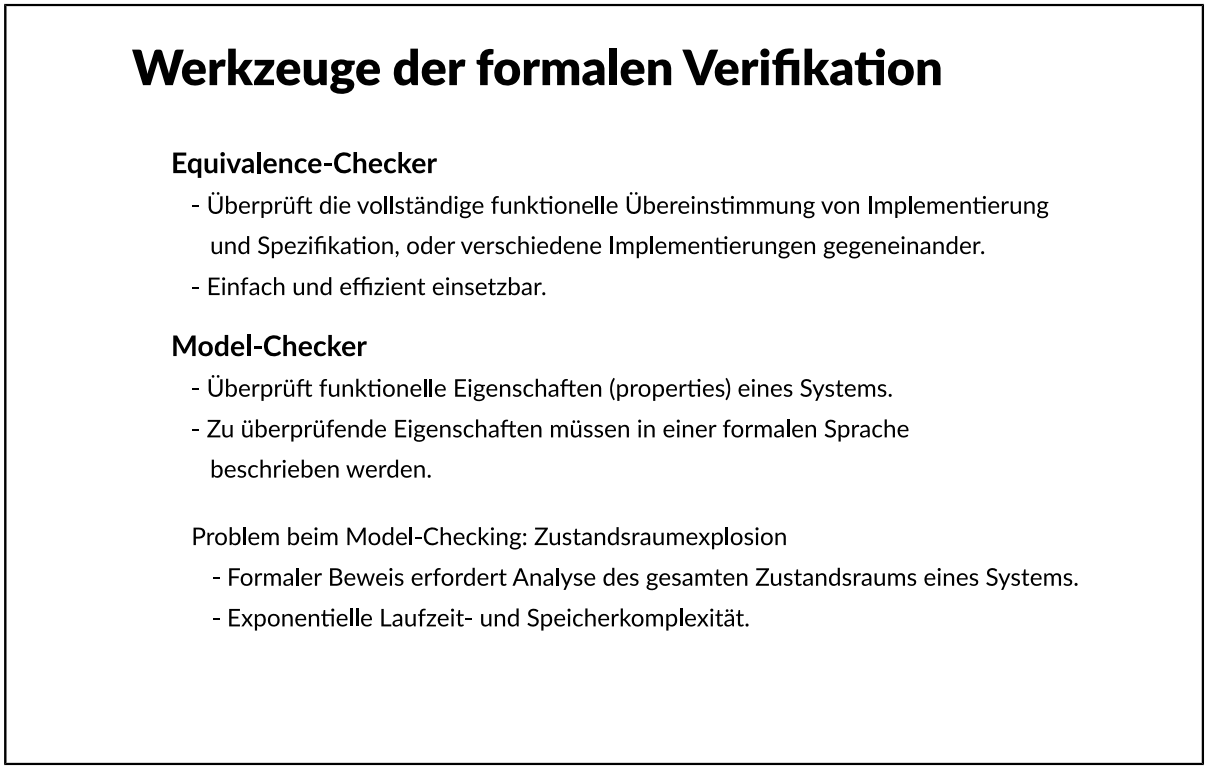
Voraussetzung für die erfolgreiche Anwendung von formalen Methoden zur Entwurfsverifikation ist eine formale Beschreibung des Testobjekts. Schaltungsdarstellungen in einer Hardware-Beschreibungssprache, aber auch strukturelle Darstellungen in Form von Netzlisten, genügen diesen Anforderungen und können somit mit formalen Methoden verifiziert werden.

Formale Verifikation: Simulation und formale Verifikation



Im Gegensatz zur Simulation, die aufgrund der endlichen Anzahl von Stimuli stets experimentellen Charakter hat, liefern die Methoden der formalen Verifikation einen mathematischen Beweis. Auf diese Weise können nicht nur Fehler im Entwurf gefunden, sondern es kann auch die Fehlerfreiheit eines Entwurfs bewiesen werden.

Formale Verifikation: Werkzeuge der formalen Verifikation



Die wesentlichen heute gebräuchlichen formalen Verifikations-Werkzeuge können in zwei Gruppen eingeteilt werden:

Equivalence-Checker:

Diese Verfahren weisen die vollständige funktionale Übereinstimmung zwischen zwei unterschiedlichen Implementierungen nach. Eingesetzt wird Equivalence- Checking z.B. um das funktionelle Modell einer Schaltung mit der Gatternetzliste zu vergleichen und somit die Korrektheit des Synthese-Schritts im Entwurfsablauf einer integrierten Schaltung formal zu beweisen. Zeitraubende Simulationen der Gatternetzliste werden damit eingespart.

Die Verfahren des Equivalence-Checkings sind weit entwickelt. Für kombinatorische Systemteile lassen sich diese Verfahren sehr einfach und effizient einsetzen.

Model-Checker:

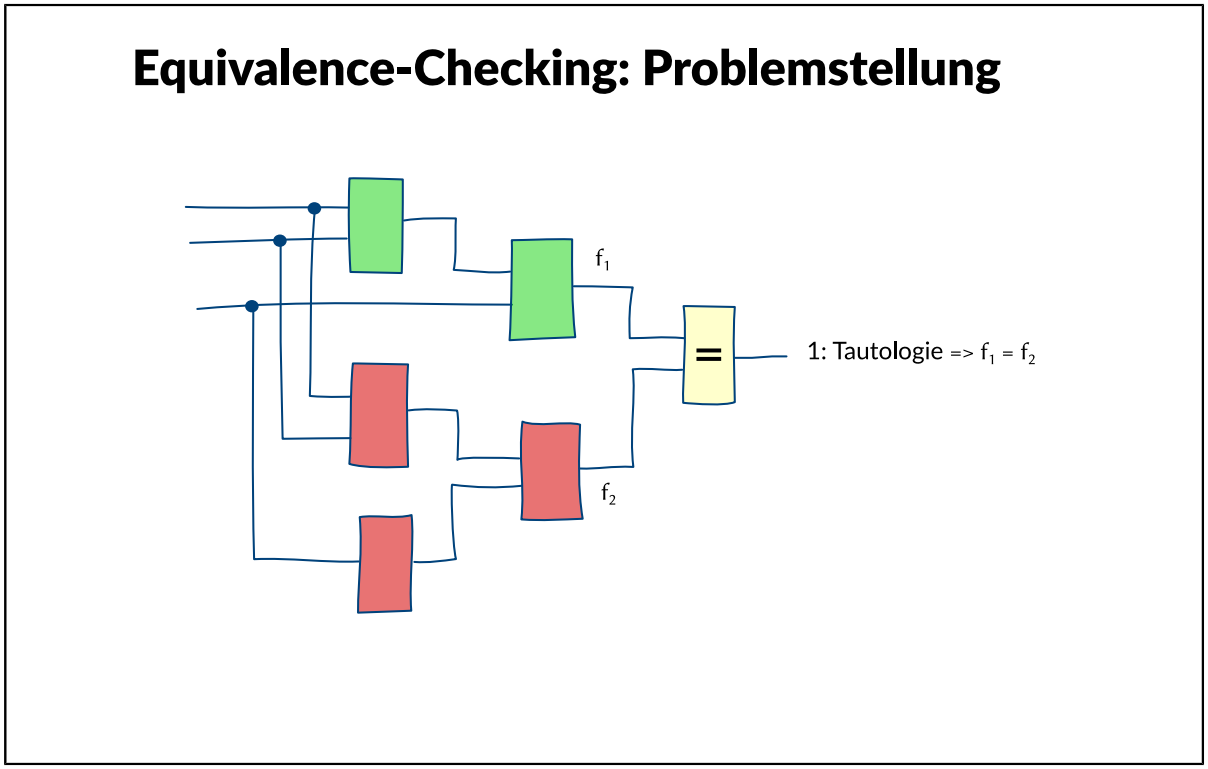
Model-Checker überprüfen einzelne Eigenschaften einer Schaltungsbeschreibung. Es kann beispielsweise zu überprüfen sein, ob eine Schaltung nach einer gewissen Zeit immer wieder in ihren Ruhezustand zurückkehrt. Die Schwierigkeit besteht darin, die zu überprüfenden Eigenschaften der Schaltung in der Eingabesprache für den Model-Checker zu beschreiben.

Generell gilt für Verfahren der formalen Verifikation, dass sie für den Anwender, den Schaltungsentwickler, schwerer zugänglich und ungewohnter sind als Simulatoren. Die notwendige Formalisierung der Eingaben stellt neue Anforderungen an den Design-Prozess und damit an den Entwickler. Auch die Ausgaben der formalen Verifikationswerkzeuge sind nicht immer leicht verständlich.

Die Grundidee des Modell-Checking ist eine Erreichbarkeitsanalyse im Zustandsraum. Damit liegt das Hauptproblem der formalen Verifikationsverfahren in der so genannten Zustandsexplosion (State Explosion). Um einen Beweis durchführen zu können, muss unter Umständen der gesamte Zustandsraum eines Systems untersucht werden. Die Zahl der zu untersuchenden Zustände steigt exponentiell mit der Anzahl der Zustandsraumdimensionen an. Die Speicher- und Laufzeitkomplexität

solcher Verfahren sind damit exponentiell in den Dimensionen des Zustandsraumes. Die exponentielle Komplexität stellt de facto eine obere Grenze für die behandelbaren Systemgrößen dar. Allerdings konnte durch verschiedene Optimierungen die Einsetzbarkeit der Verfahren erheblich gesteigert werden, so dass heute große digitale Blöcke handhabbar sind.

Formale Verifikation: Equivalence-Checking: Problemstellung



Die Grundaufgabe des Equivalence-Checking besteht darin, zwei boolesche Funktionen auf funktionelle Gleichheit zu überprüfen. Beim expliziten Ansatz betrachtet man die Verknüpfung beider Funktionen. Die beiden Ausgangssignale müssen für beliebige Eingangskombinationen stets gleich sein. Ist dies erfüllt, so sind die beiden Funktionen funktionell identisch. Dies ist im Bild durch die Verknüpfung von f_1 und f_2 über ein Äquivalenzgatter dargestellt.

Ein geeignetes Verifikationsverfahren ergibt sich aus der Idee des Widerspruchsbeweises. Nehmen wir an, die beiden Funktionen seien nicht identisch. Dann muss es eine Eingangskombination geben, für die am Ausgang des Äquivalenzgatters eine logische 0 auftritt. Umgekehrt gilt, dass, wenn es unter der Annahme eines solchen Wertes gelingt, eine gültige Eingangskombination zu finden, die Schaltungen offenbar nicht identisch sind. Gelingt es nicht, eine solche Kombination zu finden, sind die beiden Funktionen identisch. Diese Fragestellung führt unmittelbar auf das sogenannte Erfüllbarkeitsproblem.

Formale Verifikation: Erfüllbarkeitsproblem(Satisfiability,SAT)

Erfüllbarkeitsproblem (Satisfiability, SAT)

- SAT1: Überprüfung der 1-Erfüllbarkeit von booleschen Funktionen, die in konjunktiver Normalform gegeben sind
- SAT0: Überprüfung der 0-Erfüllbarkeit von booleschen Funktionen, die in disjunktiver Normalform gegeben sind

Das Finden einer Eingangskombination für eine boolesche Gleichung bei gegebenem Ausgangswert ist als Erfüllbarkeitsproblem (Satisfiability-Problem) bekannt. Dieses Problem tritt in vielen Bereichen der Informatik auf.

Formale Verifikation: SAT1-Problem

SAT1- Problem

- Geg.: Boolesche Funktion f in n Variablen $x=(x_1, x_2, .., x_n)$
  in konjunktiver Normalform
- Beispiel: $f(x) = (x_1 + \overline{x}_2 + ... + x_n) * ... * (\overline{x}_1 + x_2 + ... + \overline{x}_n)$
- Ges.: Belegung $x \in \{0,1\}^n$ mit $f(x)=1$, falls diese existiert
- Im schlechtesten Fall müssen alle möglichen Belegungen getestet werden.
- Anzahl der möglichen Belegungen: 2^n
  $n = 10$ ergibt $2^{10} = 1024 \approx 10^3$
  $n = 1000$ ergibt $2^{1000} = 2^{10*100} \approx 10^{3*100} = 10^{300}$
- Rechenzeitkomplexität von "Satisfiability": $O(2^n)$
  Satisfiability ist NP-vollständig.

SAT ist ein klassisches NP-vollständiges Problem und ist im Allgemeinen nicht effizient lösbar. Dennoch gibt es Verfahren, die in vielen praktischen Anwendungen gute Ergebnisse zeigen. Vor allem in den letzten 10 Jahren wurden neue Techniken entwickelt, die ein effizientes Lösen von Formeln mit einer großen Zahl von Variablen ermöglichen.

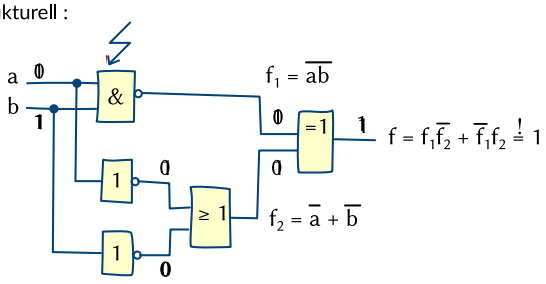
Formale Verifikation: SAT-Equivalence-Checker

SAT-Equivalence-Checker

Statt eine Tautologie $f = 1$ zu prüfen, wird ein
EXOR-Gatter genommen und nachgewiesen, dass $f = 1$ nicht erfüllbar ist,
d.h. zu einem Widerspruch führt.

Beispiel :

strukturell :



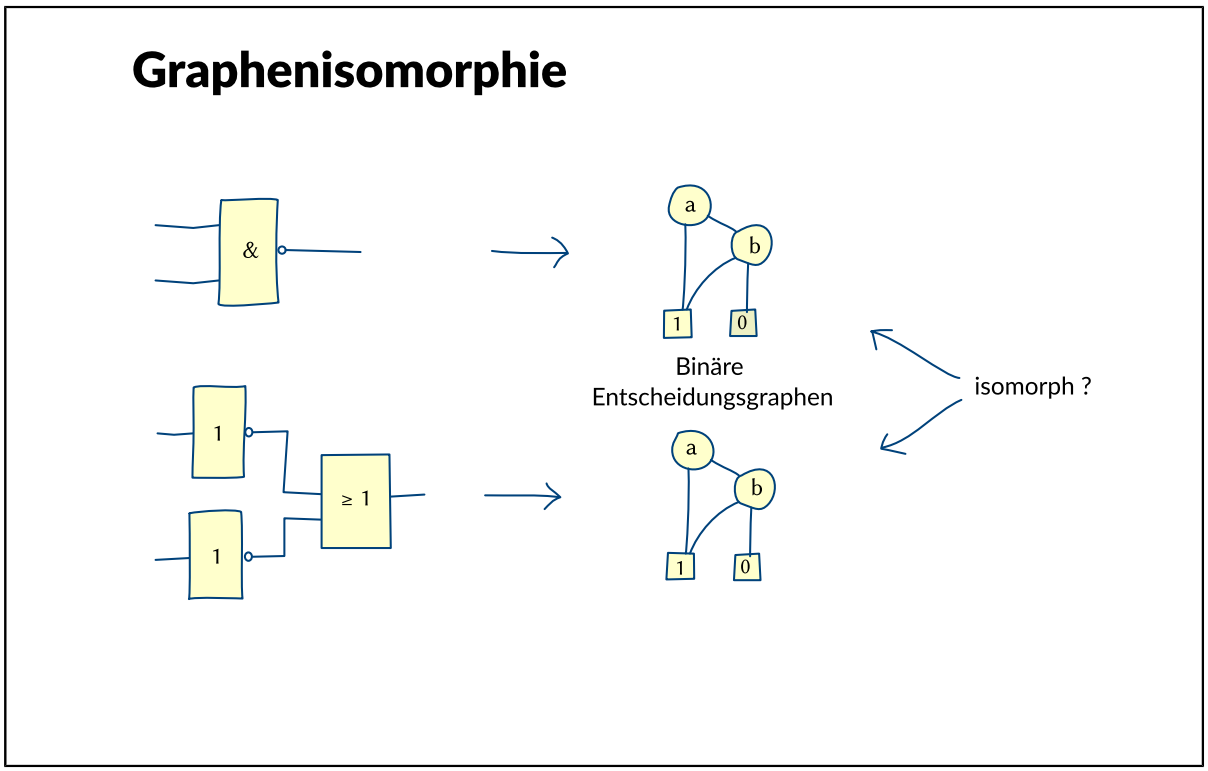
symbolisch :

$$\overline{a\,b}\,(\overline{a} + \overline{b}) + \overline{\overline{a}\,\overline{b}}\,(\overline{a} + \overline{b}) \stackrel{!}{=} 1$$
$$(\overline{a} + \overline{b})\,(a\,b) + a\,b\,(\overline{a} + \overline{b}) \stackrel{!}{=} 1$$
$$\overline{a}ab + \overline{b}ab + ab\overline{a} + ab\overline{b} \stackrel{!}{=} 1$$
$$0 \neq 1$$

Im obigen Beispiel soll die Übereinstimmung der booleschen Funktionen f_1 und f_2 geprüft werden. Wie die Rechnung zeigt, kann es nicht gelingen, für a,b eine Wertebelegung zu finden, für die sich $f=1$ ergibt, da der Ausdruck $f_1\overline{f}_2+\overline{f}_1f_2$ stets 0 ist. Das kann mit einem normalen SAT-1 Löser gemacht werden.

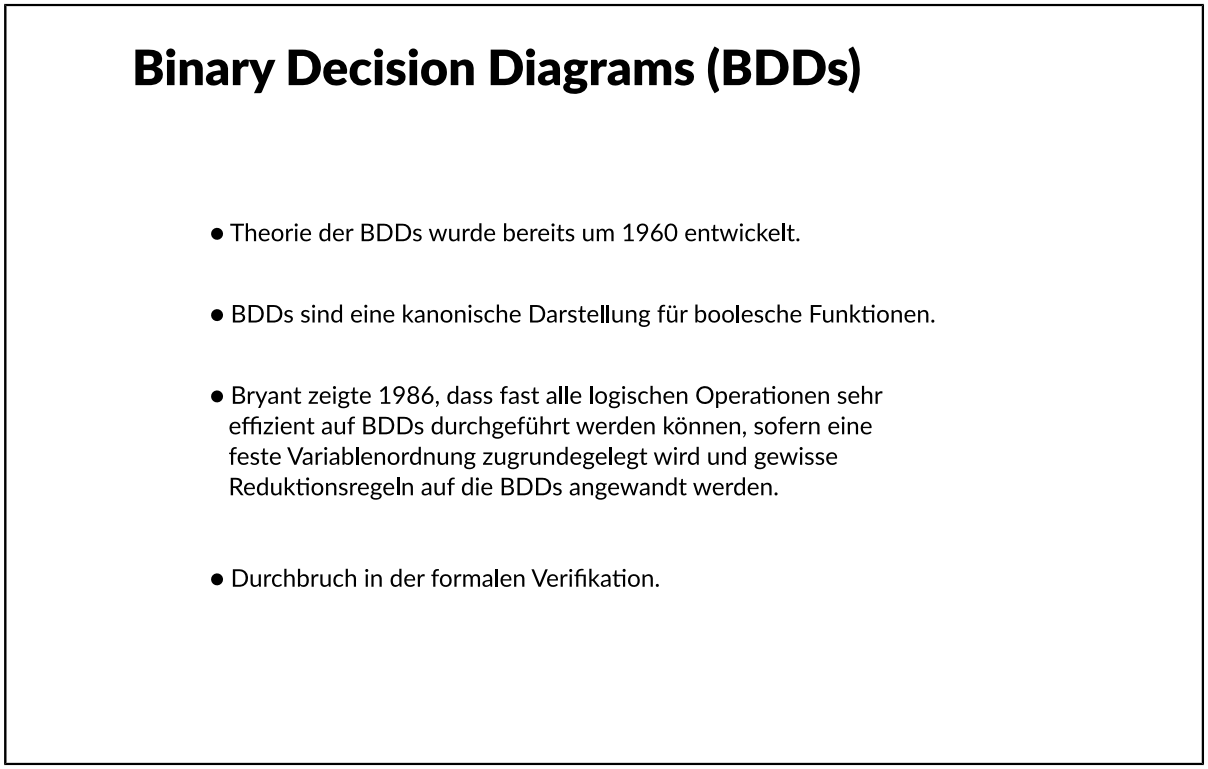
In den entsprechenden Verifikationswerkzeugen findet der Widerspruchsbeweis auch auf andere Weise statt. Grundsätzlich geeignet sind SAT-Löser, BDD-basierte Löser und auch der D-Algorithmus, der im Abschnitt Test erläutert wird. Hier werden Signale vom Ausgang zu den Eingängen propagiert, bis sich entweder eine gültige Eingangsbelegung oder ein Widerspruch ergibt.

Formale Verifikation: Graphenisomorphie



Statt ein Erfüllbarkeitsproblem zu untersuchen, kann man beim Equivalence Checking versuchen, die beiden Systeme direkt miteinander zu vergleichen. Dazu bildet man die Systeme jeweils auf einen Graphen ab und versucht, die Isomorphie der beiden Graphen zu beweisen. Damit dies möglich ist (das generelle Graphenisomorphieproblem ist NP-vollständig), muss eine eindeutige (kanonische) Darstellung gefunden werden. Diese ist durch so genannte binäre Entscheidungsgraphen Binary Decision Diagrams (BDDs) gegeben.

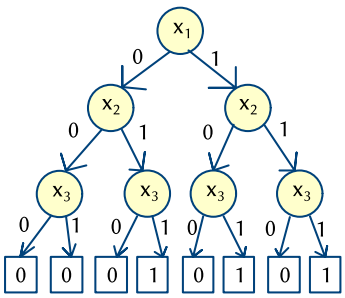
Formale Verifikation: Binary Decision Diagrams (BDDs)



Binäre Entscheidungsgraphen wurden durch R. E. Bryant entscheidend weiterentwickelt. Mit dieser Art von Graphen lassen sich boolesche Funktionen in einer sowohl kanonischen als auch kompakten Darstellung speichern und verarbeiten.

Formale Verifikation: Binärer Entscheidungsbaum: Beispiel

Binärer Entscheidungsbaum: Beispiel

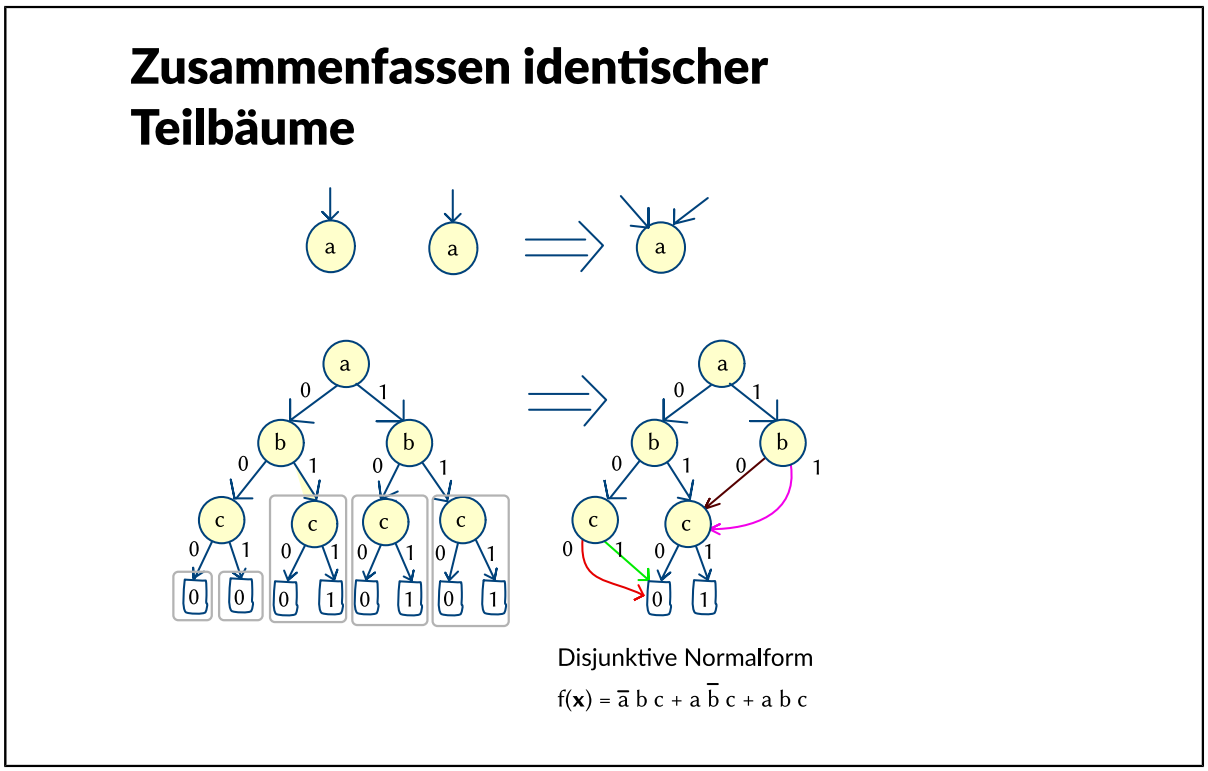


x_1	x_2	x_3	$f(\mathbf{x})$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Das Prinzip der Entscheidungsgraphen ist einfach: Von jedem Knoten des gerichteten Graphen gehen genau zwei gerichtete Kanten aus. Dem Knoten zugeordnet ist eine boolesche Variable, deren Wertebelegung mit 0 oder 1 durch die beiden Kanten repräsentiert wird. Nach dem booleschen Entwicklungssatz kann die Ausgangsfunktion durch diese Festlegung in zwei Teilfunktionen zerlegt werden. Wendet man dieses Prinzip für alle Eingangsvariablen an, so ergeben sich lediglich die beiden Senken 0 und 1 im Graphen. Der gerichtete Graph ist zyklenfrei. Durch einen Pfad wird eine Wertebelegung der Variablen eindeutig festgelegt. Die entsprechende Senke des Pfades weist dieser Wertebelegung einen Wahrheitswert zu.

Endet maximal eine Kante in jedem Knoten des Graphen, so entsteht ein binärer Entscheidungsbaum. Ordnet man alle Knoten, die derselben Variablen zugeordnet sind, auf gleicher Höhe an, so ergibt sich optisch eine übersichtliche Darstellung.

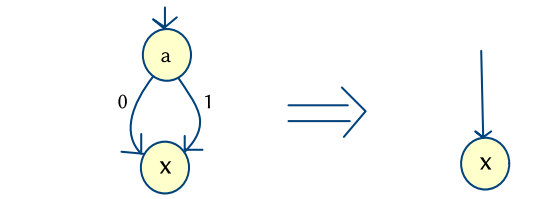
Formale Verifikation: Zusammenfassen identischer Teilbäume

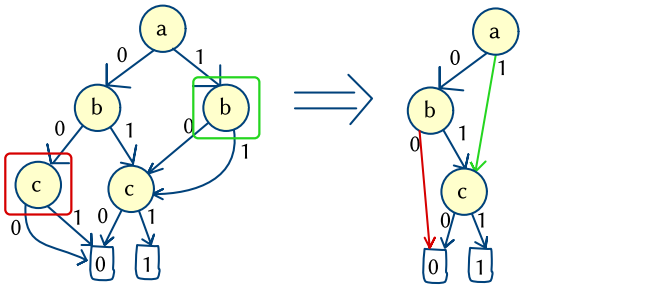


Ein binärer Entscheidungsbaum kann durch zwei Vereinfachungsoperationen vereinfacht werden. Erstens können identische Teilbäume zusammengefasst werden und zweitens können Knoten, von denen beide Kanten auf den selben Folgeknoten zeigen, entfernt werden. Führt man darüber hinaus eine feste Reihenfolge für die booleschen Variablen im Entscheidungsgraphen ein, so entsteht ein geordneter Entscheidungsgraph (Ordered Binary Decision Diagram, OBDD). Der OBDD bildet eine kanonische Darstellung einer booleschen Funktion, die gegenüber den bekannten konjunktiven und disjunktiven kanonischen Normalformen oft kompakter und leichter zu manipulieren ist.

Formale Verifikation: Entfernen überflüssiger Knoten

Entfernen überflüssiger Knoten





OBDD ROBDD

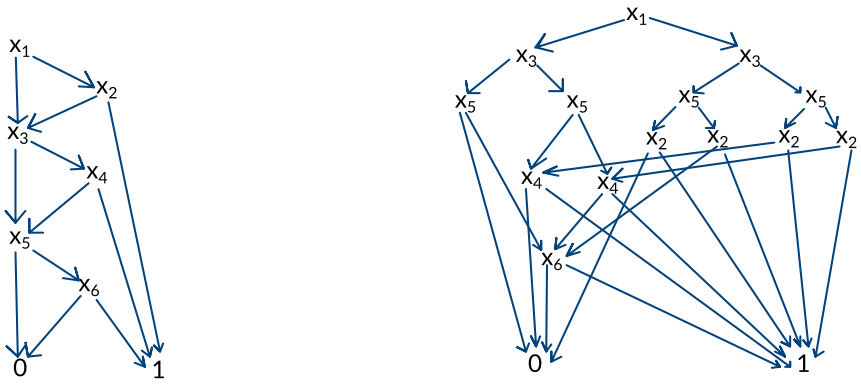
Äquivalente boolesche Funktion:

$f(\mathbf{x}) = c\,(\overline{a}\,b + a)$

Formale Verifikation: Abhängigkeit der Knotenanzahl von der Entwicklungsreihenfolge

Abhängigkeit der Knotenanzahl von der Entwicklungsreihenfolge

$$f(\mathbf{x}) = x_1x_2 + x_3x_4 + x_5x_6$$

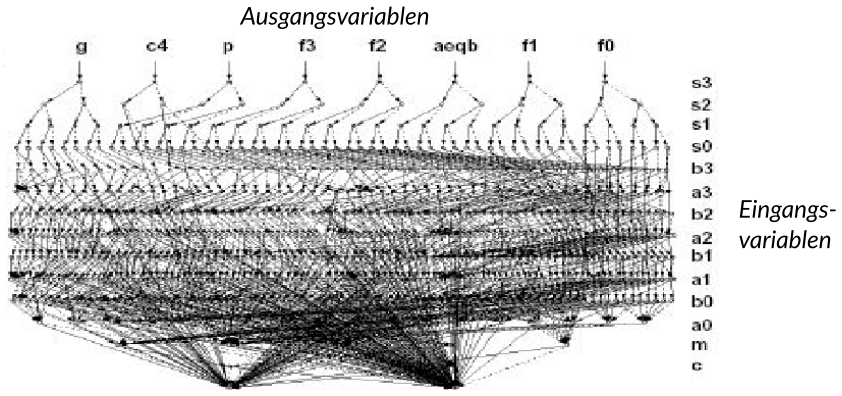


Die Wahl der Entwicklungsreihenfolge der einzelnen Variablen beeinflusst in kritischer Weise die Anzahl der Knoten des Graphen und damit den zur Äquivalenzprüfung benötigten Aufwand. Ihr ist also besondere Aufmerksamkeit zu widmen. Das Bild zeigt ein klassisches Beispiel, das 1986 von Bryant vergestellt wurde.

Formale Verifikation: Vorteile der BDDs

Vorteile der BDDs

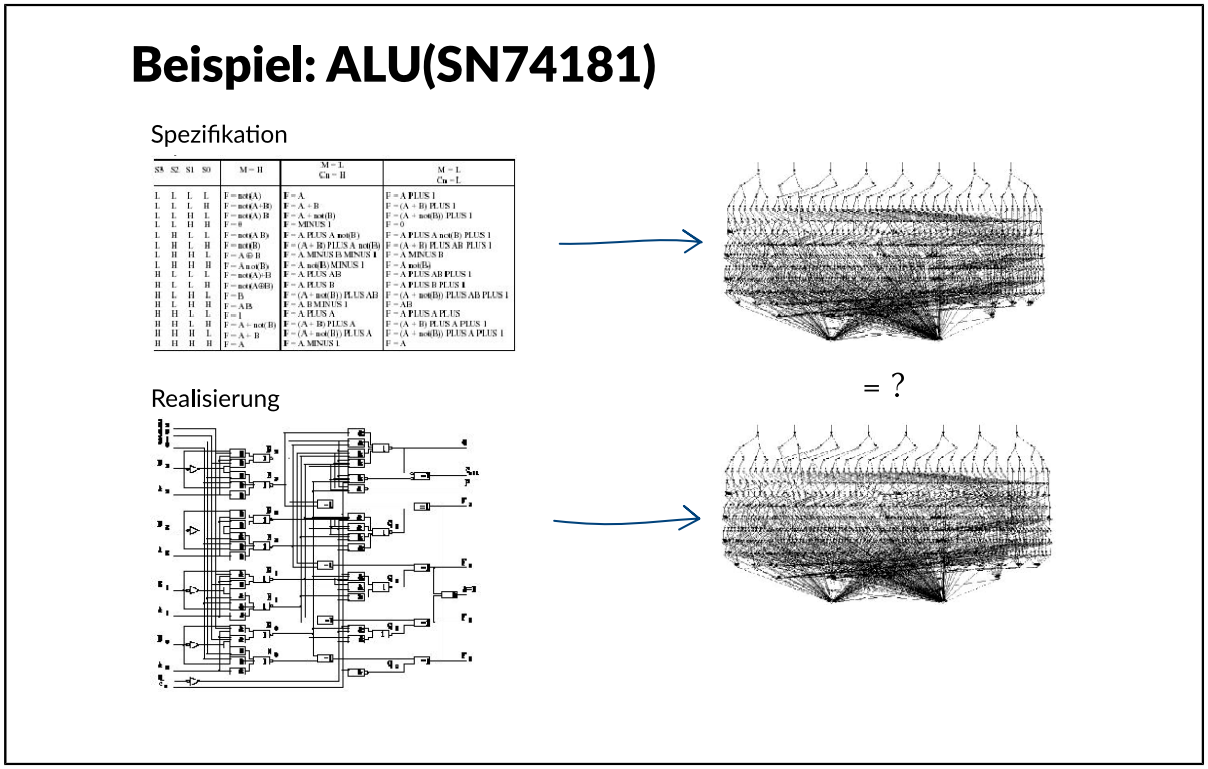
- Logische Funktionen können durch BDDs repräsentiert und mit linearer Komplexität in Zeit und Speicher aufgebaut werden.
- Äquivalenztest kann in linearer Zeit durchgeführt werden.
- Beispiel : "Shared" OBDD einer 4-Bit-ALU(SN74181)



Digitale kombinatorische Systemteile lassen sich offensichtlich mit Hilfe von OBDDs beschreiben. Die dem System zu Grunde liegende booleschen Gleichungen werden in entsprechende BDDs abgebildet. Damit ist das digitale verzögerungsfreie Verhalten des Systems vollständig wiedergegeben. Für die Konstruktion der booleschen Gleichungen aus einer Netzliste heraus gibt es zwei Ansätze. Zum einen können die Funktionen ausgehend von allen Systemeingängen aufgestellt werden, zum anderen kann die Konstruktion aber auch von den Ausgängen her erfolgen. Das Ergebnis beider Verfahren ist das gleiche, lediglich die Laufzeit der beiden Verfahren kann sich für verschiedene Systeme unterscheiden.

Das Bild zeigt beispielhaft die "Shared" OBDD einer 4-Bit-ALU (SN74181). "Shared" bedeutet, dass alle (horizontal dargestellten) Ausgangsvariablen in einem gemeinsamen Baum dargestellt werden.

Formale Verifikation: Beispiel:ALU(SN74181)



Das gezeigte Bild gibt einen Eindruck, wie zwei Systeme, die in unterschiedlicher Darstellung vorliegen (Spezifikation in funktioneller Sicht, Realisierung in struktureller Sicht jeweils auf Gatterebene) über die Beschreibungsform BDD miteinander verglichen werden können.

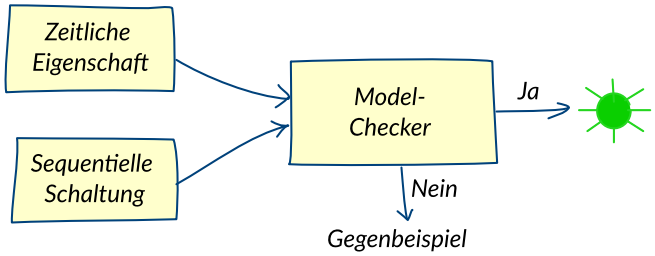
Wichtig ist, dass zum Beweis ihrer Übereinstimmung keinerlei Stimuli benötigt werden, man spricht deshalb beim Equivalence Checking auch von einer statischen Verifikationsmethode.

Für sequentielle Systemteile ist das Equivalence Checking deutlich schwieriger. In der Praxis kann es nur für Sonderfälle oder für kleine Beispiele verwendet werden. In diesem Script wird daher darauf auch nicht eingegangen.

Formale Verifikation: Model-Checking

Model-Checking

- Überprüfung zeitlicher Eigenschaften von sequentiellen Schaltungen.
- Bsp.: "Es kann nicht vorkommen, dass die Ampelsteuerung alle Signale gleichzeitig auf grün setzt."
- "Irgendwann wird jedes Ampellicht grün."



Kann die Eigenschaft nicht verifiziert werden, erzeugt der Model-Checker Informationen (Stimuli etc.) für ein entsprechendes Gegenbeispiel.

Model-Checking überprüft zeitliche Eigenschaften sequentieller Schaltungen. Das Anwendungsgebiet von Model-Checking-Verfahren ist vielfältig und reicht von der Software-Verifikation und der Verifikation von Übertragungs- Protokollen bis hin zur Überprüfung von digitalen Systemen. Entsprechend vielseitig sind auch die verwendeten Ansätze und Algorithmen. Die Grundlagen für das Model-Checking kommen aus der Informatik, wo die Verfahren zur Verifikation von Software genutzt werden. Die hier vorgestellten Algorithmen beziehen sich jedoch ausschließlich auf die Verifikation digitaler Systeme.

Beim Model-Checking werden einzelne Eigenschaften eines Systems untersucht. Dies setzt voraus, dass solche Eigenschaften in einer formalen Sprache beschrieben werden können. Je nach Art der Eigenschaft werden verschiedene Anforderungen an eine solche Sprache und damit auch an die Überprüfungsalgorithmen gestellt.

Formale Verifikation: Symbolic Model Checking

Symbolic Model Checking: Charakteristische Funktion

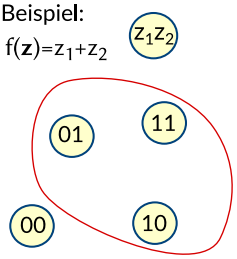
- Gegeben: Variablenvektor **z** = Zustandsvektor
- Eine Boolesche Funktion $f_A(\mathbf{z})$ kann eine Menge von Zuständen beschreiben:

$$f_A(\mathbf{z}) = \begin{cases} 1 & \text{falls } \mathbf{z} \in A \\ 0 & \text{falls } \mathbf{z} \notin A \end{cases}$$

- Sie heißt charakteristische Funktion
- Dann ist die beschriebene Zustandsmenge:

$$A = \{ \mathbf{z} \mid f_A(\mathbf{z}) = 1 \}.$$

Beispiel:
$f(\mathbf{z})=z_1+z_2$



Man erhält folgende Vorteile:
- Symbolische Beschreibung einer Zustandsmenge.
- Damit können u. U. sehr große Zustandsmengen mit einer kleinen Funktion beschrieben werden.
- Dies lässt sich auch durch BDDs realisieren.
- Übergangsfunktionen lassen sich ebenfalls so beschreiben.

Da bei sequentiellen Schaltungen (Automaten) sehr viele Zustände auftreten können, mussten Methoden gefunden werden, über Zustandsmengen mit einer großen Kardinalität zu argumentieren. Ein Durchbruch dazu war das Symbolic Model Checking. Im Wesentlichen werden dazu Beschreibungen für Zustandsmengen und Zustandsübergangsrelationen benötigt. Im Folgenden wird eine Speicherung dieser Daten mithilfe von charakterischen Funktionen erläutert.

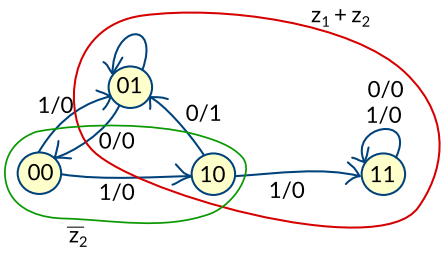
Betrachtet man alle Wertebelegungen des Variablenvektors z einer Booleschen Funktion fA, bei der das Ergebnis der Funktion 1 entspricht, so ergibt sich eine Menge von Werten A = {z | fA(z) = 1}. Identifiziert man einen Zustand eines Zustandsautomaten mit jeweils einem dieser Vektoren, können Zustandsmengen durch boolesche Funktionen und damit auch mit OBDDs beschrieben werden. Die boolesche Funktion fA(z) wird als charakteristische Funktion der Wertemenge A bezeichnet.

fA(z) = 1, falls z Element von A oder 0, falls nicht

Zur Beschreibung von Zustandsübergängen muss die Darstellung erweitert werden. Für jeden Zustand z1 bestimmt man einen oder mehrere Folgezustände z2. Das heißt, die Zustandsübergangs-Relation kann beispielsweise als delta(z1,z2) beschrieben werden. Damit können Algorithmen symbolisch über Millionen von Zuständen und deren sequentielle Abarbeitung argumentieren, ohne diese explizit aufzählen zu müssen.

Darstellung von Zustandsmengen

- Ausgangspunkt: Zustandsdiagramm einer sequentiellen Schaltung
- Boolesche Ausdrücke (charakteristische Funktionen) charakterisieren Mengen von Zuständen
- Beispiel: zwei Zustandsvariablen z_1, z_2
  - $z_1 + z_2$ charakterisiert die Menge {01, 10, 11}
  - $\overline{z_2}$ charakterisiert die Menge {00, 10}

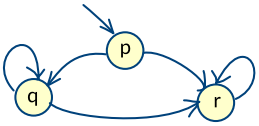


Formale Verifikation: Temporale Logik

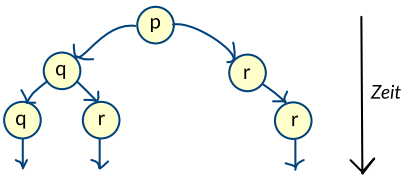
Temporale Logik

Zur Beschreibung von logischen Abläufen in der Zeit setzt man temporale Logik ein.

Zustandsübergänge werden in einen unendlichen Baum ("Berechnungsbaum") entflochten.



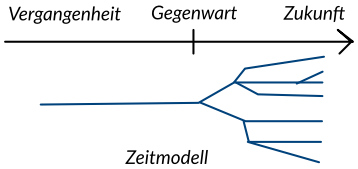
Zustandsübergangsdiagramm



Berechnungsbaum

Über die Pfade des Berechnungsbaums lassen sich Aussagen treffen - mit der Computation Tree Logic (CTL).

Das Zeitmodell lässt Verzweigungen in der Zukunft zu.



Zeitmodell

Für einfache Bedingungen in digitalen Systemen ist die boolesche Logik ausreichend. Beispielsweise könnte y = x1x2 eine zu überprüfende Spezifikationsbedingung sein. Bei genauerer Betrachtung fällt auf, dass für sequenzielle Systeme eine solche Bedingung eine unvollständige Beschreibung liefert. Soll diese Bedingung immer gelten oder ist es ausreichend, dass y = x1x2 irgendwann einmal gilt? Das heißt, für eine genauere Beschreibung sind zeitliche Zusammenhänge mit zu spezifizieren.

Dies kann durch temporale Logiken geschehen, in denen der Wahrheitsgehalt einer Funktion von der Zeit abhängt. Temporale Logiken werden in die beiden Klassen lineare temporale Logik (Linear Time Logic) und verzweigende temporale Logik (Branching Time Logic) eingeteilt. Bei den linearen temporalen Logiken geht man davon aus, dass das Systemverhalten in der Zukunft eindeutig bestimmt ist, dass es also keine verschiedenen Entwicklungen in der Zukunft geben kann. Dagegen lässt das verzweigende Zeitmodell in der Zukunft verschiedene Entwicklungen zu, die durch nichtdeterministisches Verhalten des Systems oder durch unterschiedliche Eingangsgrößen hervorgerufen werden können.

Bei dem verzweigenden Zeitmodell besteht die zeitliche Entwicklung nicht aus einer linearen Sequenz von Zuständen, sondern die Systemzustände fächern sich in der Zukunft auf. Es entsteht ein Baum, dessen Wurzel der momentane Systemzustand ist.

Demnach können zu einem Zeitpunkt in der Zukunft verschiedene Systemzustände existieren. Welchen Weg das System einschlagen wird, ist für die Betrachtung irrelevant, da alle Wege gleichberechtigt betrachtet werden, um eine allgemeingültige Aussage zu erhalten.

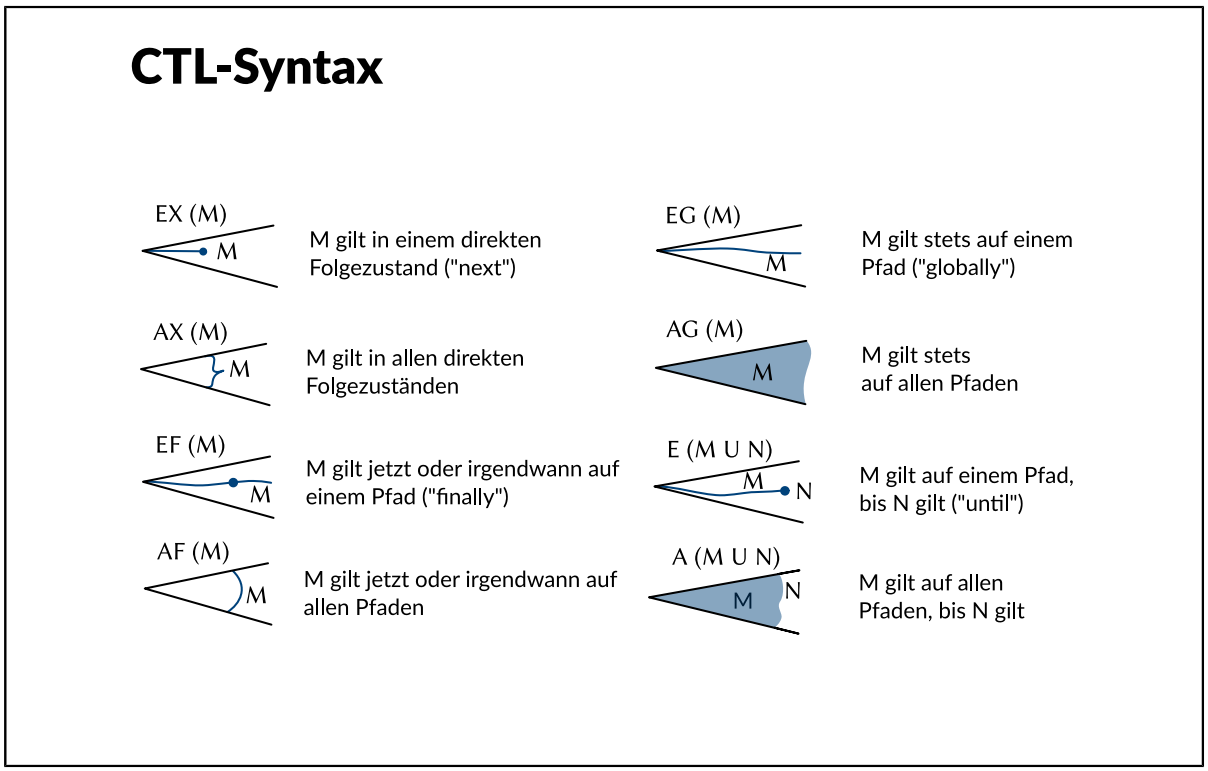
Formale Verifikation: Elemente von CTL

Elemente von CTL

Boolesche Operatoren	V = oder
	^ = und
	¬ = nicht
Pfadquantoren	A = auf allen Pfaden
	E = auf mindestens einem Pfad
Temporale Operatoren	X = im nächsten Taktschritt (Next)
	F = irgendwann (**F**inally)
	G = immer (**G**lobally)
	U = bis (**U**ntil)

Die in der Tabelle beschriebene temporale Logik wird CTL (Computeation Tree Logic) genannt. Sie geht auf einen Artikel von Clarke und Emerson aus dem Jahr 1981 zurück. Sie ergibt sich aus der booleschen Logik kombiniert mit vier temporalen Operatoren. Die vier Operatoren stehen für die Aussagen "im nächsten Taktschritt", "immer", "irgendwann" und "bis". Sie werden mit den Symbolen X, G, F und U abgekürzt. Die ersten drei temporalen Operatoren sind unär. Für die "bis"-Operation werden zwei Argumente benötigt. Der Einfachheit halber sind hier und im Folgenden bei den booleschen Funktionen nur "und", "oder" und "Negation" aufgeführt. Zusätzlich muss bestimmt werden, ob eine Bedingung auf einem Weg oder auf allen möglichen Wegen erfüllt sein muss. Dies wird durch zwei zusätzliche Pfadquantoren A und E geleistet. Eine vollständige CTL-Formel setzt sich aus jeweils einem Pfadquantor, einem temporalen Operator und einer bzw. zwei Argumentmengen zusammen.

Formale Verifikation: CTL-Syntax

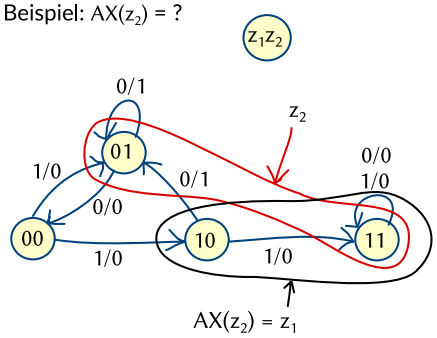


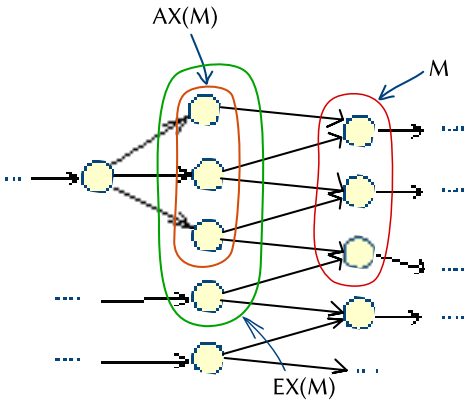
Aus der Kombination eines Pfadquantors mit einem temporalen Operator ergeben sich die acht dargestellten Ausdrücke, die jeweils auf Mengen oder auf durch boolesche Operatoren miteinander verknüpfte Mengen angewendet werden. Die Mengen beschreiben Mengen von Zuständen im Zustandsraum. Insofern befasst sich das Model-Checking mit der Erreichbarkeitsanalyse im Zustandsraum.

Formale Verifikation: AX und EX

AX und EX

- AX(M) charakterisiert alle Zustände, die nur Nachfolgezustände haben, die durch M charakterisiert sind.
- EX(M) charakterisiert alle Zustände, von denen aus wenigstens ein Nachfolgezustand durch M charakterisiert ist.



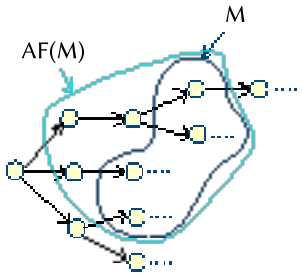


Ausschnitt aus dem Computation Tree

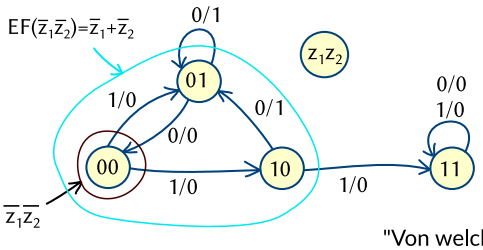
Formale Verifikation: AF and EF

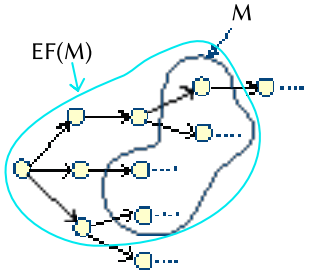
AF und EF

- AF(M): Zustände, von denen aus M unvermeidbar ist
  AF(M) charakterisiert alle Zustände, von denen aus M irgendwann sicher eintritt (Lebendigkeitseigenschaft), z.B. irgendwann zeigt die Ampel grün.
- EF(M): M ist erreichbar (jetzt oder später)



Beispiel: $EF(\overline{z}_1\overline{z}_2) = ?$





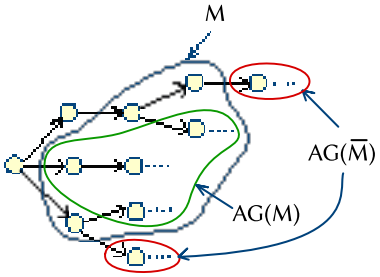
"Von welchen Zuständen aus ist es möglich, wieder in den Anfangszustand zu kommen?"

Formale Verifikation: AG und EG

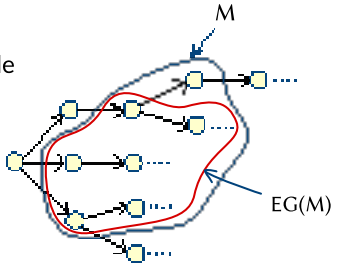
AG und EG

- AG(M): Alle aktuellen und alle Nachfolgezustände sind durch M charakterisiert.

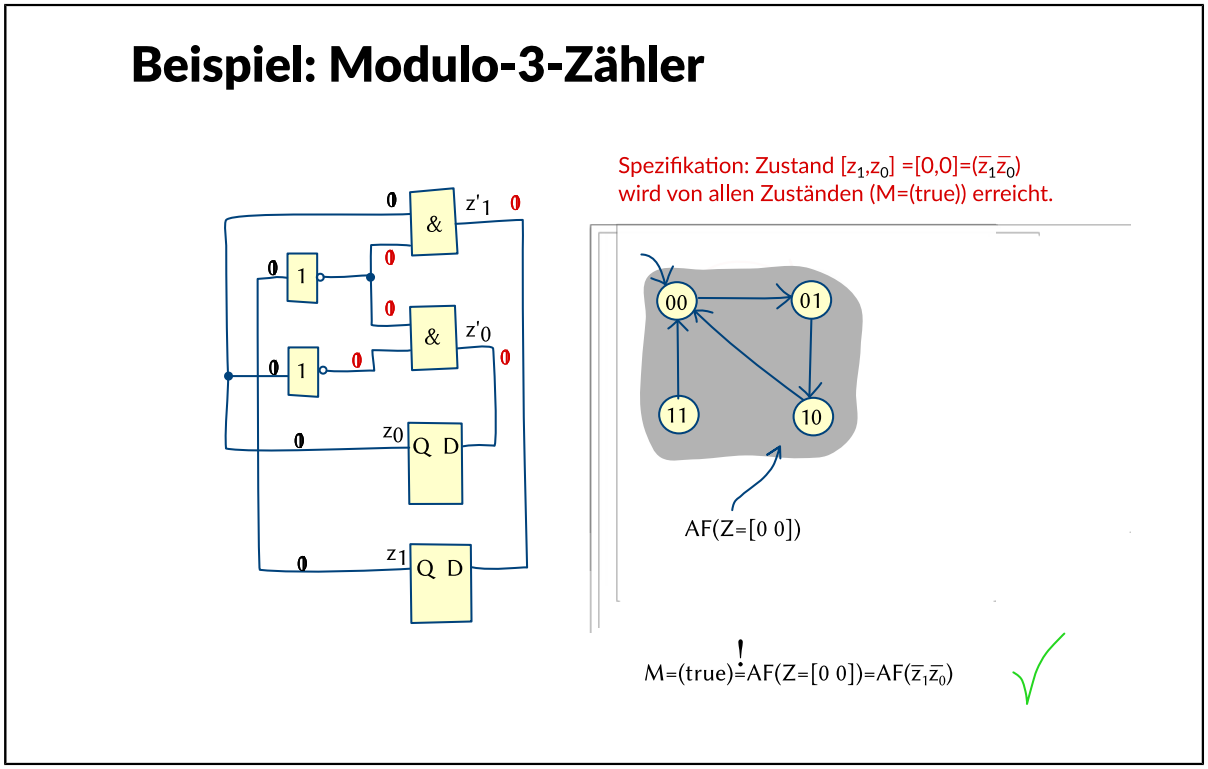
  $AG(\overline{M})$ charakterisiert alle Zustände, von denen aus M nie eintreten kann (Sicherheitseigenschaft) z.B. es tritt nie ein, dass alle Ampeln grün sind.



- EG(M): Es gibt mindestens einen Pfad, auf dem der aktuelle und alle Nachfolgezustände durch M charakterisiert sind.



Formale Verifikation: Beispiel: Modulo-3-Zähler



In diesem Beispiel wird beispielhaft eine Eigenschaft eines Modulo 3 Zählers überprüft. Zunächst muss dazu die Gatterschaltung in das Zustandsübergangsdiagramm umgewandelt werden. Dies geschieht im realen, OBDD-basierten Modelchecker jedoch nur implizit durch Aufstellen der OBDDs. Der Check-Prozess ist hier animiert dargestellt und liefert ein positives Ergebnis für die spezifizierte Eigenschaft, d.h. der Zustand [0,0] wird tatsächlich immer (d.h. von jedem anderen Zustand) erreicht.